GoLang Recon Automation Script

A comprehensive, fully-commented GoLang script that automates bug bounty reconnaissance using industry-standard tools and integrates seamlessly with the Caido proxy for advanced security testing.

Overview

This script orchestrates multiple reconnaissance tools to discover critical attack surface information across your target domain. It provides comprehensive visibility into potential security vulnerabilities through automated discovery processes.

Subdomains

Using subfinder, assetfinder, and amass

Live Hosts

Validated with httpx probing

Open Ports

Comprehensive nmap scanning

Hidden Directories

Fuzzing with ffuf

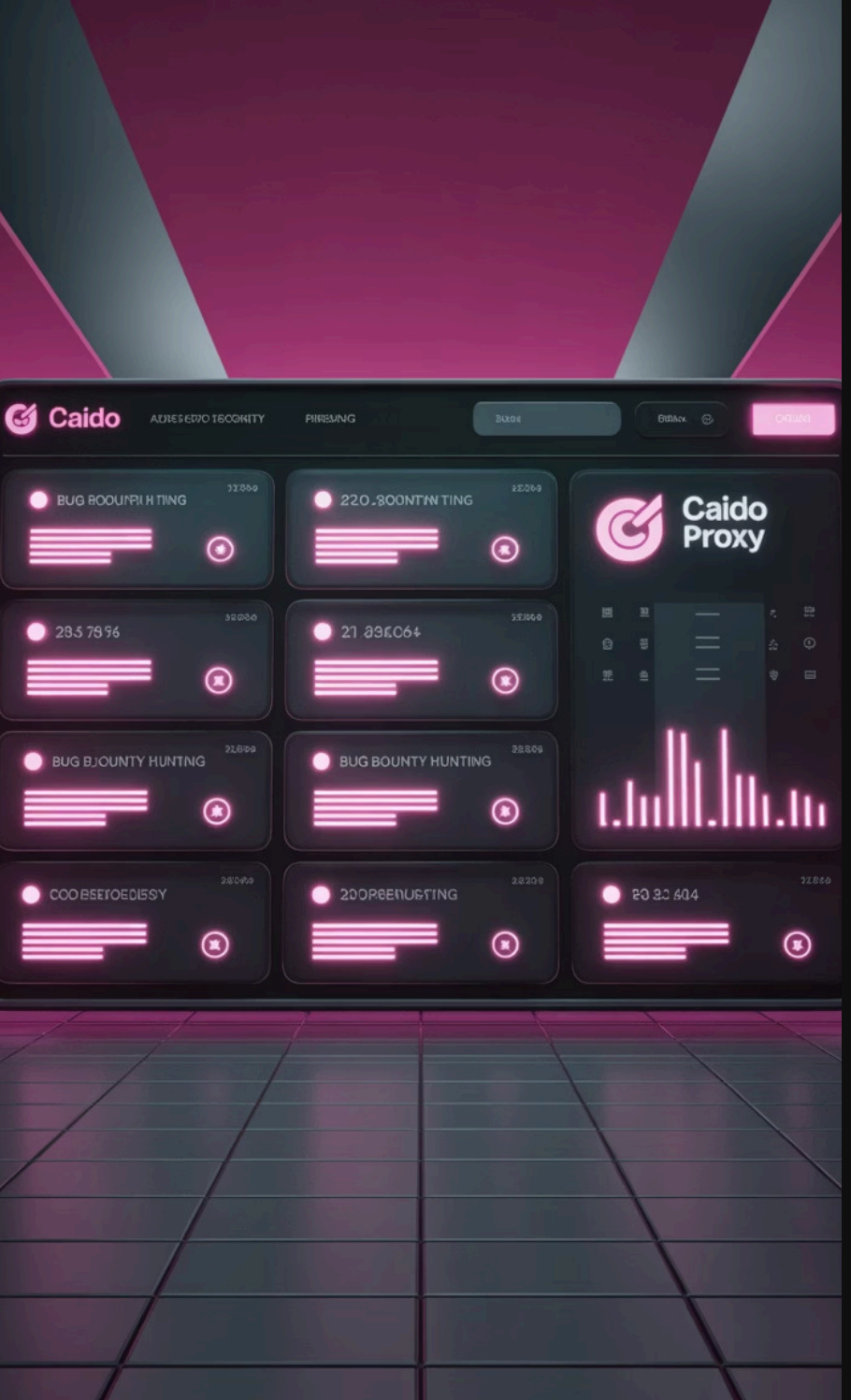
JS Endpoints

Extracting API endpoints

Hidden Parameters

Discovery using Arjun

All HTTP traffic is routed through Caido for later manual testing and analysis, ensuring comprehensive visibility into all reconnaissance activities.



Key Features

Parallel Execution

Multiple tools run simultaneously for faster scanning, reducing total reconnaissance time by up to 3x compared to sequential execution.

Caido Integration

All HTTP traffic routes through Caido proxy, enabling comprehensive manual testing and analysis of discovered endpoints.

Organized Output

Results stored in timestamped directories with clear file naming conventions for easy navigation and analysis.

JSON Results

Machine-readable output format enables easy parsing, integration with other tools, and automated post-processing workflows.

Error Handling

Comprehensive error handling ensures the script continues running even if individual tools fail, maximizing data collection.

Progress Logging

Real-time status updates throughout the process keep you informed of scan progress and tool execution status.

The Complete Code: Core Structure

The script begins with essential imports and data structures that form the foundation of the reconnaissance automation. These components handle everything from file operations to concurrent execution.

```
package main
import (
  "bufio"          // Buffered reading/writing - improves I/O performance
  "encoding/json"   // JSON marshaling/unmarshaling - data serialization
  "fmt"            // Formatted I/O operations
  "log"            // Logging with timestamps
  "os"             // Operating system operations
  "os/exec"         // Executing external commands
  "path/filepath"   // Cross-platform file path manipulation
  "strings"         // String manipulation
  "sync"           // Synchronization primitives for goroutines
  "time"           // Timestamps and time operations
)
```

ReconConfig Structure

Centralizes all configuration settings including target domain, output directory, wordlist path, Caido proxy URL, thread count, and scan type flags. This structure makes the script highly customizable.

ReconResults Structure

Organizes all findings from the reconnaissance process including subdomains, live hosts, directories, parameters, JS endpoints, and nmap results. Designed for easy JSON export.

Main Function: Program Entry Point

The main function orchestrates the entire reconnaissance workflow, validating inputs, initializing configuration, creating output directories, and executing each reconnaissance phase in sequence.

01

Validate Arguments

Checks command-line arguments to ensure a target domain is provided before proceeding with any initialization.

02

Initialize Configuration

Sets up ReconConfig with sensible defaults including target domain, timestamped output directory, wordlist path, Caido proxy URL, thread count, and scan flags.

03

Create Output Directory

Uses os.MkdirAll to create the output directory structure with 0755 permissions, ensuring all parent directories exist.

04

Initialize Results

Creates ReconResults structure with initialized maps for storing discoveries from each reconnaissance phase.

```
config := ReconConfig{
 Target: os.Args[1],
 OutputDir: fmt.Sprintf("recon_%s_%s", os.Args[1], time.Now().Format("20060102_150405")),
 Wordlist: "/usr/share/wordlists/SecLists/Discovery/Web-Content/common.txt",
 CaidoProxyURL: "http://127.0.0.1:8080",
 Threads: 50,
 UsePassiveScan: true,
 UseActiveScan: true,
}
```

Phase 1: Subdomain Enumeration

The first reconnaissance phase discovers all subdomains to expand the attack surface. Multiple tools run in parallel, each querying different data sources including passive DNS, certificate transparency logs, Facebook, VirusTotal, and Archive.org.

Subfinder

Queries passive DNS and certificate transparency logs for comprehensive subdomain discovery.

Assetfinder

Searches Facebook, VirusTotal, Archive.org, and additional sources for subdomain records.

Amass

OWASP project providing comprehensive passive enumeration across numerous data sources.

> **Why multiple tools?** Each tool has different data sources and discovery techniques. Running them in parallel saves time while maximizing subdomain discovery coverage.

The function uses goroutines for parallel execution, with a WaitGroup ensuring all tools complete before proceeding. A mutex protects the shared subdomain slice from race conditions. Results are deduplicated since different tools often find the same subdomains.

Phase 2: HTTP Probing

This phase validates which discovered subdomains actually host active web services. Not all DNS records point to live servers, so HTTP probing is essential for identifying real targets.

The runHTTPProbing function uses httpx to send HTTP/HTTPS requests to all discovered subdomains. It writes subdomains to a temporary file for efficient processing, then executes httpx with flags for status codes, page titles, and technology detection.

The function parses httpx output to extract live host URLs, filtering for successful responses. This creates a focused list of active targets for subsequent reconnaissance phases.

```
output, err := runCommand("httpx",
  "-l", tempFile,
  "-silent",
  "-status-code",
  "-title",
  "-tech-detect")
```

200

OK Status

Successful responses

301

Redirects

Moved permanently

403

Forbidden

Auth required

Phase 3: Port Scanning

Port scanning discovers open ports and services beyond HTTP/HTTPS. Services running on non-standard ports may reveal additional vulnerabilities and attack vectors. This phase only executes if active scanning is enabled and live hosts were discovered.

Semaphore Pattern

Buffered channel limits concurrent operations to 5 simultaneous nmap scans, preventing network congestion and avoiding IDS/IPS triggers.

Aggressive Timing

Uses -T4 timing template for faster scans while scanning the top 1000 most common ports for comprehensive coverage.

Individual Results

Each scan result is saved to its own file with sanitized hostname, making it easy to review port scan findings per host.

```
output, err := runCommand("nmap",
  "-Pn",            // Skip host discovery
  "-T4",            // Aggressive timing
  "--top-ports", "1000",
  cleanHost)
```

The function uses goroutines with a semaphore to control concurrency. Each scan cleans the host URL to extract just the hostname, runs nmap with appropriate flags, and stores results in a thread-safe manner using a mutex.

Phase 4: Content Discovery

Content discovery finds hidden directories, files, and endpoints not linked in the application. This phase uses ffuf to fuzz URLs with a wordlist, discovering resources like admin panels, backup files, and configuration endpoints.

Fuzzing Strategy

The function limits processing to the first 20 hosts to keep runtime reasonable, as content discovery can take hours on large wordlists. It uses a semaphore to limit concurrent fuzzing operations to 10, balancing speed with resource management.

Each fuzzing operation creates a unique output filename and runs ffuf with carefully chosen flags for status code matching, thread count, JSON output format, and **critical Caido proxy routing**.

Caido Integration

The -x flag routes ALL traffic through the Caido proxy at http://127.0.0.1:8080. This ensures every discovered endpoint appears in Caido's HTTP history for later manual testing and analysis.

Results are saved in JSON format for easy parsing, with the parseFFUFOutput function extracting discovered URLs from the structured output.

```
_, err := runCommand("ffuf",
 "-u", fmt.Sprintf("%s/FUZZ", h),
 "-w", config.Wordlist,
 "-mc", "200,204,301,302,307,401,403",
 "-t", fmt.Sprintf("%d", config.Threads),
 "-o", outFile,
 "-of", "json",
 "-x", config.CaidoProxyURL, // Routes through Caido
 "-silent")
```

Phase 5: JavaScript Analysis

JavaScript files often contain undocumented API endpoints, API keys, secrets, and sensitive data. This phase extracts JavaScript file URLs from web pages for further analysis.

API Endpoints

JS files frequently contain hardcoded API endpoints not documented in public-facing interfaces.

Secrets & Keys

Developers sometimes accidentally commit API keys, tokens, and credentials in JavaScript code.

Sensitive Data

Client-side code may expose internal URLs, usernames, or configuration details useful for reconnaissance.

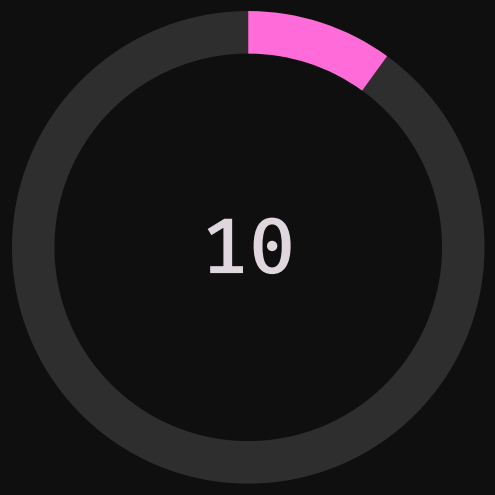
The runJSAnalysis function uses a simple but effective approach with curl and grep. It downloads each page and uses Perl regex to extract JavaScript file references from src attributes. The regex pattern uses positive lookbehind and lookahead to precisely match .js file URLs.

```
curl -s %s | grep -oP '(?<=src=")[^"]*\.js(?=")' | head -20
```

Results are limited to 20 hosts and 20 JS files per host to keep runtime manageable. More sophisticated tools like LinkFinder, JSParser, or JSFScan could be integrated for deeper analysis.

Phase 6: Parameter Discovery

Hidden or undocumented parameters are common sources of vulnerabilities including IDOR (Insecure Direct Object References), injection attacks, and authentication bypasses. This phase uses Arjun to test for common parameter names.



Host Limit

Maximum hosts tested due to slow execution time

Parameters

Arjun tests thousands of parameter names per URL

The function limits processing to 10 hosts as parameter discovery is extremely time-intensive. Arjun runs with the --stable flag for more reliable but slower detection. The --fast flag could be used for quicker but less thorough scans.

```
output, err := runCommand("arjun",
  "-u", host,
  "--stable")
```

Discovered parameters are stored in the results map and can reveal hidden functionality, debug endpoints, or vulnerable input vectors that weren't obvious from normal application usage.

Results Export & Summary

After all reconnaissance phases complete, the script exports comprehensive results in multiple formats and displays summary statistics. This ensures findings are accessible for both automated processing and manual review.

JSON Export

Complete results structure exported to results.json with pretty-printed formatting for easy parsing and integration with other tools.

Summary Statistics

Console output displays total counts of discovered subdomains and live hosts, providing immediate feedback on reconnaissance success.

1 2 3

Individual Files

Each reconnaissance phase saves its findings to dedicated text files: subdomains.txt, live_hosts.txt, nmap results, ffuf JSON, JS files, and parameter lists.

```
log.Printf("\n[✓] Reconnaissance complete! Results saved to: %s\n", config.OutputDir)
log.Printf("[*] Found %d subdomains, %d live hosts\n", len(results.Subdomains), len(results.LiveHosts))
```

The saveResultsJSON function uses json.MarshalIndent to create human-readable JSON with 2-space indentation. Files are written with 0644 permissions (owner read/write, others read-only) for appropriate security.

Helper Functions: Core Utilities

The script includes several essential helper functions that handle command execution, file operations, data deduplication, and output parsing. These utilities ensure robust operation and clean data handling throughout the reconnaissance process.

runCommand

Executes external commands and returns combined stdout/stderr output. Captures both streams to ensure errors are visible alongside normal output.

saveToFile

Writes string slices to files with buffered I/O for better performance. Skips empty lines and ensures proper file closure even on errors.

deduplicate

Removes duplicate strings using a map for O(1) lookups. Essential since multiple tools often discover the same subdomains.

sanitizeFilename

Makes URLs safe for filenames by removing protocols, slashes, and colons. Prevents filesystem errors from invalid characters.

parseFFUFOutput

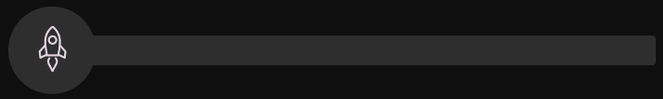
Extracts URLs from ffuf's JSON output format. Reads the entire file, unmarshals JSON, and returns a simple list of discovered URLs.

```
func deduplicate(slice []string) []string {
  keys := make(map[string]bool)
  result := []string{}
  for _, item := range slice {
    if item != "" && !keys[item] {
      keys[item] = true
      result = append(result, item)
    }
  }
  return result
}
```

```
func sanitizeFilename(s string) string {
 s = strings.ReplaceAll(s, "http://", "")
 s = strings.ReplaceAll(s, "https://", "")
 s = strings.ReplaceAll(s, "/", "_")
 s = strings.ReplaceAll(s, ":", "_")
 return s
}
```

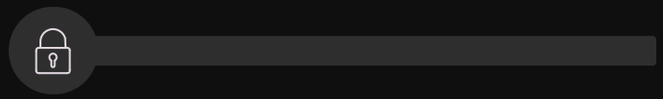
Subdomain Enumeration Deep Dive

The runSubdomainEnum function demonstrates sophisticated concurrent programming patterns in Go. It launches three reconnaissance tools simultaneously using goroutines, with careful synchronization to prevent race conditions.



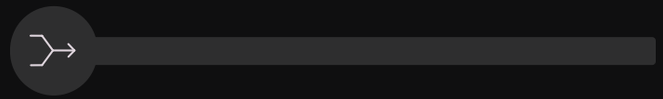
Launch Goroutines

Each tool runs in its own goroutine (lightweight thread), allowing all three to execute simultaneously. The WaitGroup counter tracks active goroutines.



Mutex Protection

A mutex protects the shared allSubdomains slice from race conditions. Only one goroutine can modify the slice at a time, preventing data corruption.



Combine Results

After all goroutines complete, results are deduplicated since different tools often find the same subdomains through different data sources.

```
for _, tool := range tools {
  wg.Add(1)
  go func(t struct { name string; command []string }) {
    defer wg.Done()
    output, err := runCommand(t.command...)
    mu.Lock()
    allSubdomains = append(allSubdomains, subs...)
    mu.Unlock()
  }(tool)
}
wg.Wait()
```

This parallel execution pattern reduces total runtime by approximately 3x compared to sequential execution, while maintaining thread safety through proper synchronization primitives.

Port Scanning & Content Discovery

Both port scanning and content discovery use semaphore patterns to control concurrency. This prevents overwhelming the network or target systems while maintaining reasonable scan speeds.

Port Scanning Strategy

The runNmapScan function uses a buffered channel as a semaphore with capacity 5, limiting concurrent nmap scans. Each goroutine acquires a slot before scanning and releases it when done, allowing the next scan to proceed.

This prevents network congestion and avoids triggering intrusion detection systems. The -T4 timing template provides aggressive but controlled scanning of the top 1000 most common ports.

```
semaphore := make(chan struct{}, 5)
semaphore <- struct{}{}        // Acquire
defer func() { <-semaphore }() // Release
```

Content Discovery Approach

The runContentDiscovery function limits concurrent fuzzing to 10 operations and processes only the first 20 hosts. Content discovery is extremely time-intensive, so these limits keep runtime manageable.

Each fuzzing operation routes traffic through Caido using the -x flag, ensuring all discovered endpoints appear in Caido's HTTP history for manual testing. Results are saved in JSON format for easy parsing.

```
semaphore := make(chan struct{}, 10)
_, err := runCommand("ffuf",
 "-x", config.CaidoProxyURL)
```

JavaScript & Parameter Analysis

The final reconnaissance phases focus on extracting hidden functionality from JavaScript files and discovering undocumented parameters. These techniques often reveal vulnerabilities not obvious from normal application usage.

JavaScript Extraction

Uses curl and grep with Perl regex to extract .js file URLs from HTML source. The regex pattern (?<=src=")[^"]*\.js(?=") uses positive lookbehind and lookahead for precise matching. Results are limited to 20 files per host to keep runtime reasonable.

Parameter Discovery

Arjun tests thousands of common parameter names against each URL using the --stable flag for reliable detection. This is extremely slow, so processing is limited to 10 hosts. Discovered parameters often reveal hidden functionality or vulnerable input vectors.

Advanced Analysis

More sophisticated tools like LinkFinder, JSParser, or JSFScan could be integrated for deeper JavaScript analysis. These tools can extract API endpoints, secrets, and sensitive data from minified or obfuscated code.

Both functions save their findings to dedicated files for later manual review. JavaScript files should be examined for hardcoded credentials, API keys, internal URLs, and undocumented endpoints. Parameters should be tested for injection vulnerabilities, IDOR, and authentication bypasses.

Installation & Prerequisites

Before running the reconnaissance script, you must install Go and all required security tools. This section provides complete installation commands for Ubuntu/Debian systems.

01

Install Go

Install the Go programming language runtime required to execute the script.

```
sudo apt install golang-go
```

02

Install Go Tools

Install reconnaissance tools written in Go using go install commands.

```
go install -v github.com/projectdiscovery/subfinder/v2/cmd/subfinder@latest
go install -v github.com/tomnomnom/assetfinder@latest
go install -v github.com/projectdiscovery/httpx/cmd/httpx@latest
go install -v github.com/ffuf/ffuf@latest
```

03

Install System Tools

Install amass and nmap from system repositories.

```
sudo apt install amass nmap
```

04

Install Python Tools

Install Arjun parameter discovery tool using pip3.

```
pip3 install arjun
```

05

Configure PATH

Add Go binaries directory to your PATH environment variable.

```
echo 'export PATH=$PATH:~/go/bin' >> ~/.bashrc
source ~/.bashrc
```

SecLists Wordlists

Install the comprehensive SecLists wordlist collection for content discovery:

```
sudo apt install seclists
# Or manually:
git clone https://github.com/danielmiessler/SecLists.git /usr/share/wordlists/SecLists
```

Usage & Output Structure

Basic Usage

Run the script directly with Go or build an executable for repeated use:

```
# Direct execution
go run recon.go target.com

# Build executable
go build -o recon recon.go
./recon target.com
```

Output Structure

The script creates a timestamped directory containing all reconnaissance results:

```
recon_target.com_20251208_015030/
  ├─── subdomains.txt
  ├─── live_hosts.txt
  ├─── nmap_target_com.txt
  ├─── ffuf_target_com.json
  ├─── js_target_com.txt
  └─── results.json
```

subdomains.txt

Complete list of all discovered subdomains from subfinder, assetfinder, and amass.

live_hosts.txt

Filtered list of subdomains with active web servers responding to HTTP/HTTPS requests.

nmap_*.txt

Individual port scan results for each host showing open ports and detected services.

ffuf_*.json

Content discovery results in JSON format with discovered directories and files per host.

js_*.txt

Extracted JavaScript file URLs for each host, useful for finding API endpoints and secrets.

results.json

Complete reconnaissance results in structured JSON format for automated processing.

Caido Integration & Customization

All HTTP traffic from the content discovery phase routes through Caido at http://127.0.0.1:8080, enabling comprehensive manual testing of discovered endpoints.

Using with Caido

- Start Caido application
- Ensure proxy is listening on 127.0.0.1:8080
- Run the reconnaissance script
- All discovered endpoints appear in Caido's HTTP history
- Use Caido's Replay feature to manually test interesting endpoints

Customization Options

Modify the ReconConfig struct in main() to customize behavior:

```
config := ReconConfig{
  Target: os.Args[1],
  Wordlist: "/path/to/wordlist.txt",
  CaidoProxyURL: "http://127.0.0.1:8080",
  Threads: 100,
  UsePassiveScan: true,
  UseActiveScan: false,
}
```

Error Handling

The script includes comprehensive error handling: tool failures are logged but don't stop execution, empty results are checked before proceeding, all file operations are validated, and missing tools are clearly indicated.

Performance Considerations

Parallel execution runs 3 subdomain tools simultaneously. Semaphores limit concurrent operations (5 nmap scans, 10 fuzzing operations). Host limits keep runtime reasonable (20 for content discovery, 10 for parameters). Buffered I/O improves file performance.

Legal Usage & Next Steps

⚠️ CRITICAL: Legal & Ethical Usage

Only run this script on targets you have explicit permission to test:

- Bug bounty programs with defined scope
- Your own infrastructure and systems
- Authorized penetration tests with written permission

Unauthorized scanning is illegal and unethical. Always obtain proper authorization before conducting reconnaissance activities.

Next Steps

- Review the results.json file for complete findings
- Check Caido's HTTP history for all discovered endpoints
- Manually test interesting endpoints in Caido
- Review JavaScript files for sensitive data
- Test discovered parameters for vulnerabilities

Troubleshooting

"command not found" errors: Ensure all tools are installed and in your PATH:

```
which subfinder httpx ffuf nmap arjun
```

Caido proxy errors: Ensure Caido is running and listening on port 8080:

```
netstat -tuln | grep 8080
```

Permission errors: Some operations may require elevated privileges:

```
sudo ./recon target.com
```